

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Assistant Commissioner for Patents
Washington, D.C. 20231

jc685 U.S. PTO

Atty Dkt.: 2452-13

Sir:



01/18/00

Date: January 18, 2000

Attached for filing is the patent application of:

Inventor: **CRUTCHER**Entitled: **APPLET EMBEDDED CROSS-PLATFORM
CACHING**

and including attachments as noted below:

☒ Declaration, ☒ Abstract27 pages of specification and claims (including 31 numbered claims), and
5 sheets of accompanying drawing/s.☐ Record & return the attached assignment to the undersigned.☐ Priority is hereby claimed under 35 USC 119 based on the following foreign applications, the entire content of which is hereby incorporated by reference in this application:

Application Number	Country	Day/Month/Year Filed
--------------------	---------	----------------------

, respectively.

☐ Certified copy(ies) of foreign application(s) is/are attached..☐ Please amend the specification by inserting before the first line --This is a _____ of PCT application _____, filed _____, the entire content of which is hereby incorporated by reference in this application.--☐ Priority is hereby claimed under 35 USC 120/365 based on the following prior PCT applications designating the U.S., the entire content of which is hereby incorporated by reference in this application:

Application Number	Country	Day/Month/Year Filed
--------------------	---------	----------------------

This application is based on the following prior provisional application(s):

Application No.	Filing Date
-----------------	-------------

respectively, the entire content of which is hereby incorporated by reference in this application, and priority is hereby claimed therefrom.

Please amend the specification by inserting before the first line: -- This application claims the benefit of U.S.

Provisional Application No. _____, filed _____, the entire content of which is hereby incorporated by reference in this application.

Verified Statement attached establishing "small entity" status (Rules 9 & 27)

The Examiner's attention is directed to the prior art cited in the parent application by applicant and/or Examiner for the reasons stated therein.

Preliminary amendment to claims (attached hereto), to be entered before calculation of the fee below.

Also attached:

FILING FEE IS BASED ON CLAIMS AS FILED LESS ANY HERewith CANCELED

Basic Filing Fee		\$	690.00
Total effective claims	31 - 20 (at least 20) = 11	x \$	18.00
		\$	198.00
Independent claims	5 - 3 (at least 3) = 2	x \$	78.00
		\$	156.00
If any proper multiple dependent claims now added for first time, add \$260.00 (ignore improper)		\$	0.00
		SUBTOTAL	\$ 1,044.00
If "small entity," then enter half (1/2) of subtotal and subtract		-\$	0.00
		SECOND SUBTOTAL	\$ 1,044.00
Assignment Recording Fee (\$40.00)		\$	0.00
		TOTAL FEE ENCLOSED	\$ 1,044.00

Any future submission requiring an extension of time is hereby stated to include a petition for such time extension.

The Commissioner is hereby authorized to charge any deficiency in the fee(s) filed, or asserted to be filed, or which should have been filed herewith (or with any paper hereafter filed in this application by this firm) to our **Account No. 14-1140**. A duplicate copy of this sheet is attached.

1100 North Glebe Road, 8th Floor
Arlington, Virginia 22201-4714
Telephone: (703) 816-4000
Facsimile: (703) 816-4100
RWF:ej

NIXON & VANDERHYE P.C.

By Atty: Robert W. Faris, Reg. No. 31,352

Signature: 

Our Ref.: 2452-13

U.S. PATENT APPLICATION

Inventor(s): Craig Crutcher

Invention: APPLLET EMBEDDED CROSS-PLATFORM CACHING

***NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100***

SPECIFICATION

APPLET EMBEDDED CROSS-PLATFORM CACHING

FIELD OF THE INVENTION

5 This invention relates to method and apparatus for efficiently caching applets on a client computer.

BACKGROUND AND SUMMARY OF THE INVENTION

10 It used to be that code to be executed on a personal computer or a workstation was usually supplied by physically inserting a diskette, optical disk or other storage medium into a local drive associated with the personal computer or workstation. Now, such code can simply be downloaded over a computer network. One of the more significant developments in network-based software downloading over the last few years has been the development of Java applets and the Java virtual machine -- which allow programs to be dynamically downloaded for execution on an as-needed basis.

15 Briefly, an applet is a small executable code module that normally doesn't have the complete features and user interface of a normal application. The applet runs inside of an application (for example, a standard web browser) within a "virtual machine" -- that is, a set of computer resources and instructions that make up a generally standardized environment for the applet's execution. Java is the language
20 most commonly associated with applets, and standard web browsers and other applications include Java-based virtual machines to run Java applets.

Such applets provide a convenient mechanism for flexibly providing client-side functionality. They can provide all sorts of functionality on the client side -- everything from graphics support to game play to database lookups to security
25 functions and more. Since Java is a general purpose language, virtually any

functionality can be expressed in an applet -- but, as discussed below, there are some practical limitations.

In general, how much functionality an applet can provide depends on the applet's size. Small applets download quickly and provide adequate response times, but are limited in terms of their functionality. Larger applets can provide broader functionality but take proportionally longer to download, load and start up. At some point, download/load/startup delay becomes a major hindrance to the use of large applets. No one wants to use an application that takes fifteen minutes to load.

Caching has been used for many years to reduce the time required to load code or data. The idea of caching applets on the client computer is not new. Startup time can be drastically reduced by caching the applet on the client machine. The overall time savings is inversely proportional to the web server connection. Assuming the connection between the client and server is the gating factor, the less information transferred from the server to the client, the better. If the applet can be cached locally the first time it is requested, the user only pays the download penalty once. Subsequent startups should be noticeably faster. Furthermore, there are other reasons besides speed performance for persistently caching applets -- for example, reducing network traffic associated with repeatedly downloading the same applet on numerous occasions to the same client machine, and the possibility of flexibly generating vendor-independent persistent class libraries

Unfortunately, caching an applet in today's web browser environments can be a daunting task. This is at least in part because each of the various web browsers and Java virtual machines protect against rogue applets by imposing security constraints and requirements on applet persistence. Currently, the primary

commercially available browsers (e.g., Netscape Communicator and Navigator, and Microsoft's Internet Explorer) and applet execution platforms each implement applet caching in different, proprietary ways. Some do not implement applet caching at all. To minimize the risk that a rogue applet will damage a client
5 computer, web browsers generally deny downloaded applets the ability to persist after the web browser has been shut down. Such security precautions if enforced will prohibit persistent applet caching altogether.

Despite a security philosophy that discourages downloaded applets from persisting, certain available applications provide a limited ability to allow applets to
10 persist. As one example, Marimba Inc.'s CASTANET™ software distribution infrastructure provides a Java application that provides deployment and local caching of both applets and applications. See for example U.S. Patent No. 5,919,247 to Van Hoff. However, the Van Hoff technique requires a separate application to be loaded to handle applet channels, and the technique is based on a
15 file updating concept requiring a particular series of identifiers and/or indices. The Van Hoff technique is not generally applicable across a wide range of different applet execution platforms, and also suffers from other disadvantages.

In contrast to such a separate-application approach, it would be highly desirable to develop applet caching capabilities that would work across a number of
20 client-side platforms such as standard web browsers available from Netscape, Microsoft and others. However, Netscape's Communicator browser, Microsoft's Internet Explorer browser in Windows and MacIntosh, and JavaSoft's web browser Plug-In have differing security models and signing mechanisms and differing caching abilities. As a result, deploying applet caching to a given platform requires
25 specific code for that platform (e.g., specific packaging and signing on both

Communicator and IE), and is not even available on some platforms (Plug-In, IE w/MRJ on Mac, HotJava, etc.). Even the html is specific to some platforms. For at least these reasons, flexible, persistent platform-independent applet caching has not been realized in the past.

5 This invention solves this problem by distributing a caching mechanism implemented by an applet -- providing a mechanism and framework for caching applets in a modular and cross-platform manner. In accordance with one aspect of the invention, an applet incorporates a lightweight caching mechanism into its root set of classes. The remainder of the applet is divided into functional modules that
10 can be subsequently downloaded as needed. The initial applet and caching mechanism are packaged and signed in a package using tools and procedures native to each platform. Each functional module of the applet can be packaged and signed in a generic, platform-independent fashion for verification and loading by any of the various platform-dependent initial applet packages. In this way, the same
15 functional module portions of the applet (which may comprise the bulk of the applet) are cached and loaded in a platform-independent manner without requiring or relying on any caching mechanism built into the platform.

 Since the initial applet package does not need to contain more than only a small number of classes (e.g., a caching mechanism and a class loader), the initial
20 download is short, and startup time is reduced.

 In accordance with another aspect of the invention, a cache manager including a class loader is constructed and initialized during applet initialization. The initial applet package makes successive calls to the cache manager, asking it to load further, platform-independent functional modules. For each load request, the
25 cache manager checks to see if the request has already been satisfied, and if not,

loads the set of classes into memory via the cache manager's classloader. If the module has not already been loaded, the local cache is checked first. If the module is found in the cache, a version check is made against the requested version. If the cached version is found to be compatible, the module is loaded from the cache.

5 Otherwise, the module is retrieved from the server, cached, and loaded.

The following is a non-exhaustive listing of further features and advantages provided by this invention:

- An applet persistent mechanism that is consistent across many supported platforms.
- Ability to download applets in a piecemeal fashion, with control over the granularity of package size so as minimize initial download size.
- A mechanism to allow and support incremental changes to applet modules.
- Applet-based deployment and caching - Caching and loading classes and modules on demand based on construction of a class loader within an applet environment in a platform independent manner.
- Cross-platform caching - Providing a platform independent caching mechanism reduces a significant amount of work that the applet developer has to do to deploy to multiple platforms. Because caching is implemented by an applet, it is not necessary to know about or use any particular caching mechanisms on the client's application (which can vary from one platform to another).
- Modular caching - Breaking an applet up into functional groups, and then fetching them on an as-needed basis, allows the applet to start up quicker by eliminating the download of unneeded classes or other parts of the

applet (e.g., fonts, images, sounds, etc.). The applet itself directs what additional modules are needed and when. Additionally, functional groups can be shared by any number of applets. One applet can benefit from another applet having already downloaded a needed group of functional modules.

- Optimized compression and storage of applet modules - Module contents are packaged in an optimal manner for downloading that maximizes the compression algorithms available. By packaging all classes and other components into one stream and compressing only the stream, compression overhead is reduced. As a result, compressed modules are considerably smaller than a compressed standard archive file with the same classes. In contrast with the standard archive (which digitally signs each individual file and stores the resulting digests as a "manifest" file in the archive), we sign the entire stream once --eliminating the need for the manifest file. This reduces overall size and increases "unpacking" efficiency by reducing sign verification overhead.
- A lightweight, wrapping applet that implements the caching mechanism can be used to launch other applets without their needing to implement cache manager. The wrapped applet can be packaged, signed, and used for deploying and caching an applet on a number of platforms without need for packaging or signing anything for each specific platform.
- Applications can be broken up into functional modules that may be versioned and distributed separately.

- Communication between the main applet classes and the functional modules is routed through a communication service that is downloaded with the main applet.

BRIEF DESCRIPTION OF THE DRAWINGS

5 These and other features and advantages provided by the present invention will be better and more completely understood by referring to the following detailed description of example embodiments in conjunction with the drawings, of which:

 Figure 1 schematically illustrates an overall applet caching technique provided by an example embodiment of this invention;

10 Figure 2 is a flowchart of exemplary computer-controlled steps;

 Figure 3 schematically illustrates example class loader and persistent caching architecture;

 Figure 4 schematically illustrates the relationship between different types of classes in the example embodiment;

15 Figure 5 shows example program controlled logic the example embodiment uses to test whether to obtain a functional module from cache or from the server; and

 Figure 6 shows an example compressed data stream.

DETAILED DESCRIPTION OF EXAMPLE EMBODIMENTS

20 Figure 1 shows an example persistent applet caching system provided in accordance with an example embodiment of this invention. The Figure 1 system 100 includes a server 102 and more or more clients 104. In this example, server 102 may be a conventional web server computer including a processor, local disk storage and an interconnection to the Internet, an Intranet or other computer and/or

digital communications network 105. Client 104 may include a conventional personal computer, workstation or other computing appliance including a local hard disk or other non-volatile store 106, a display 104a, a local processor and an interconnection to network 105.

5 Server 102 and client 104 can communication together over the network 105. In one example, server 102 comprises a web server that supplies web pages and other information, and client 104 comprises a personal computer equipped with a standard web browser such as Netscape Navigator or Communicator or Microsoft Internet Explorer, which browser has (or is extended by a plug-in to have) a so-called Java virtual machine. The client 104's web browser requests web pages from web server 102 over network 105 and displays them on a display 104a. Server 102 can also download applets to client 104 over the network 105 for execution by a virtual machine provided by client 104. Such virtual machine can be, for example, a conventional Java Virtual Machine that is now standard in most web browsers in wide use today.

15 In the example embodiment, applet 110 includes an initial applet package 110-I and one or more additional functional modules 111a, 111b, Initial applet package 110-I incorporates a light weight caching mechanism in its root set of classes. In the example embodiment, initial applet package 110-I (including this persistent caching mechanism) is packaged and signed using tools and procedures native to the particular platform (e.g., web browser) existing on client 104. The example embodiment thus provides different "flavors" of initial applet package 110-I for different platforms, for example:

- Microsoft IE 4/5 for Windows (using native VM)
- Microsoft IE for Mac (using MRJ)

- Netscape Communicator 4.06+ for Windows (using native VM)
- JavaSoft's Plug-In 1.1.2+
- JavaSoft's AppletViewer
- Others

5 The different initial applet package 110-I "flavors" or versions may, in one embodiment, be functionally identical (and may in fact be written in the same language such as Java and some of the same code) --- but differ in that they are customized for the specific security, signing, packaging (and any other) requirements the various different platforms require to verify, load and run the

10 applet. The initial applet packages 110-I are thus customized by request and obtain the privileges to perform class loading and caching from the particular platform they are written for. Such requirements are well known, and are widely available from Microsoft, Netscape, JavaSoft and others who develop virtual machines that can run applets.

15 In the example embodiment, server 102 determines which of these "flavors" of initial applet package 110-I to download to client 104 in a conventional fashion by identifying the particular application being run at the client. Server 102 may determine which platform is running on client 104 so it can download the initial applet package 110-I that is appropriate to that platform. For example, the HTML

20 may/can contain enough information to direct the client 104's browser to the correct set of classes (e.g., Microsoft Internet Explorer will favor CAB files over JAR files when given the choice); or server 102 can send an http command to the client's web browser requesting identification of the client application by name, version number, etc. Server 102 then downloads the particular "flavor" of initial applet package 110-

I that will meet the security and other requirements of client computer 104's associated browser or other applet execution platform.

Once initial applet package 110-I is downloaded, loaded and is running on client computer 104 (the executing version of this applet module is referenced by numeral 110-I'), it constructs, initializes and creates a cache manager 112 including a class loader on the client computer. The executing applet module 110-I' then makes successive calls to the cache manager 112, asking it to retrieve and load further applet functional modules 111 as they are needed. Executing module 110-I' includes a map table or other mechanism that maps commands or other service requests into requests to retrieve and load functional modules 111 capable of performing the requested service(s).

As mentioned above, the remaining classes of applet 110 are divided into functional modules (e.g., "feature A" module 111a, "feature B" module 111b, etc...) that can be platform-independent. In the example embodiment, each functional module 111a, 111b, etc., consists of a set of classes and/or other data that are packaged and signed in a way to permit it to be verified and loaded by any of the various "flavors" of initial applet package 110-I. In the example embodiment, these functional modules 111 are platform-independent in the sense that any initial applet package 110-I' (irrespective of platform) will be able to accept, verify and load them. Depending on the particular requirements, most of the functionality (and thus the length) of applet 110 may be placed within functional modules 111 -- minimizing the download time and complexity of initial applet package 110. The division of applet 110 into different functional modules may be based on functions or services to be performed, the complexity of each of the various functions, etc. If desired,

functional modules 111 for other needed services can be reused by sharing them between different applets 110.

Modules 111 may depend on other modules 111 in order to operate. For example, classes in one module 111 may need classes in another module 111 in order to load. To provide for such module dependency, the example embodiment conveys dependency information within the module 111. The module loader, while trying to load a module 111, will extract the list, and ensure that all listed modules (and their respective versions) are already loaded. If a needed module 111 is not already loaded, loading of the current module is put on hold until the required module is loaded. This process is recursive, to provide for loading dependent modules of dependent modules of dependent modules (ad infinitum).

Determining which functional modules 111 to load and when, can be based on static information and/or dynamic events, such as:

- Module list embedded/loaded with initial applet
- User input event
- System event
- Server event
- Server-generated html (the applet tag contains additional modules to load based on current context).

Figure 2 shows a flowchart of overall program control steps of a process 140 performed by the system 100 shown in Figure 1. In this example, initial applet package 110-I is stored on server 102 as a small signed compressed file (e.g., a Microsoft "cabinet" (.cab) file or a Java ARchive (JAR) file). As will be understood by those skilled in the art, the .cab and JAR file formats are conventional, standardized file compression formats that allow many individual files

to be stored and downloaded together in compressed format within a single HTTP transaction to a browser or other applet execution platform.

In this example, the small, digitally signed compressed file comprising initial applet package 110-I contains the main applet and loader classes -- including a caching mechanism used to subsequently load additional functional modules such as "feature A" module 111a, "feature B" module 111b, etc. In the example embodiment, the functional modules 111 are subsequently loaded on an as-needed basis.

The initial applet package 110-I downloaded from server 102 to client 104 over network 105 (block 150; Figure 2 path 113), and is authenticated, decompressed and executed by a standard web browser or other applet execution platform on client computer 104. Once applet 110-I is running (thus establishing cache manager 112), it may request loading of other compressed archive files (i.e., additional applet class and/or resource files) comprising functional modules 111a, 111b, etc. (decision block 152). For each load request, the cache manager 112 checks to see if the request has already been satisfied. If not, the cache manager 112 attempts to load a set of classes or resources (e.g., images, fonts, etc.) into memory via the cache manager's class loader, and the local cache 106 is checked to see if the functional module 111 to be loaded is already present in the cache (decision block 154). The decision block is downloaded with the applet and can, therefore, be tailored to suit individual application requirements that could change for a given application across time.

If the module has not already been loaded and the module is found in the cache 106, a version check is made against the requested version (decision block 154). If local cache 106 contains an appropriate version of the functional module

111 ("yes" exit to decision block 154), the functional module 111 is loaded from the cache and all of its classes are added to the class loader's namespace (Figure 2 block 156; Figure 1 path 114).

If the functional module 111 is not found in cache 106 or the cached version is old ("no" exit to decision block 154), the functional module 111 is retrieved from server 102 over network 105 (Figure 2 block 158; Figure 1 path 118). The retrieved functional module 111 may, or may not, be persisted in cache 106 dependent upon how the load call is made. In some cases, the functional module 111 retrieved from server 102 is stored to cache 106 and then loaded via the cache manager ("yes" exit to decision block 160; block 162). In other cases, the retrieved functional module 111 is not locally cached ("no" exit to decision block 160), and is simply loaded. If the retrieved functional module 111 has a dependency list, all dependent functional modules 111 are loaded prior to loading the requested functional module(s) (blocks 164, 166). In the example embodiment, recursion is used to load dependent functional modules 111 (i.e., flow returns to block 152) to accommodate arbitrary dependency nesting.

To protect against tampering, all functional modules 111 downloaded from server 102 in the example embodiment are digitally signed. The digital signature is verified at download time. Once functional module 111 has made it into cache 106 of client 104, it will be considered trusted. Trusting the contents of cache 106 is no less secure than existing implementations on commonly available web browsers such as Internet Explorer and Netscape.

Figure 3 shows an example basic structure and component communication for portions of system 100 existing on client 104 once initial applet package 110-I' has been loaded and is running. In this example, cache manager 112 communicates

with and manages non-volatile cache 106 . Cache manager 112, in turn, accepts commands from a module loader 200. Module loader (200) creates and manages a platform specific classloader via the module classloader (202). This platform specific classloader 202 manages volatile (in-memory) cache of the module classes and resources.

In the example embodiment, the applet 110 talks to module loader 200 to load modules and retrieve the specific instance of the classloader that should be used. In the example embodiment, the classloader 202 derives from the system class, java.lang.Classloader. The applet 110 uses module classloader 202 directly or indirectly to load classes from modules managed by module loader 200. The module classloader 202 can and will retrieve classes and resources from both the applet and the primordial classloader.

Module loader 200 is thus directed by applet 110 to load modules 111 via cache manager 112 and/or module class loader 202. Module class loader 202 (which may include a variety of different conventional class loaders, e.g., an MS class loader 204a, a default class loader 204b, etc. existing as part of a conventional web browser or other applet execution platform) is also present, and acts to connect one or more classes together into an executable. In this example, these various class loaders 204 can communicate (while observing appropriate namespace restrictions -- see below) with applet class loader 206 to load additional classes for execution.

In finding a class, the example embodiment module loader 200 first checks to see if the class has already been loaded into memory but has not yet been resolved. If the class has not yet been loaded by module classloader 202, then module loader 200 determines whether the primordial class loader has the class. If the class has

not yet been loaded, module loader 200 checks whether the class has been downloaded but not yet loaded. Optionally module loader 200 can also check whether the class is in an applet-supplied URL (a module loader classpath of sorts). Module loader 200 may also check whether the parent classloader has the class. In other words, the search order can be as follows to provide enhanced efficiency and security:

1. Is class already loaded?
2. Does primordial classloader have the class?
3. Do we have the class, but haven't loaded it?
4. (optional) Is it in the applet supplied URL? (a module loader classpath of sorts)
5. Does the parent classloader (normally the applet class loader 206) have the class?
6. Forget it, we give up.

Note that this load order places priority on classes in the module loader 200's immediate name space. Therefore, if the class is present in both the parent name space and the module loader 200's name space, the module loader 200's version will be returned.

Figure 4 shows that the module class loader 202's name space 250 is a superset of the name space 252 of applet class loader 206 -- which in turn is a superset of the name space 254 of system class loader 202. The Figure 4 diagram implies communication boundaries for cross-class communication. In order for two objects, each loaded from a different class loader, to communicate with each other, the class used for communication must be accessible to both objects.

Therefore, the class must be loaded via a class loader whose name space exists for both objects.

As discussed above, the module loader 200 makes a version test on every functional module 111 it reads from cache 106. Module loader 200 also tests version compatibility of functional modules 111 the first time they are downloaded from server 102. In one specific example, archive (JAR) version numbers may be represented as strings or 32-bit integers using the following format:

Format: MMmmmttbbb*

MM	Major version number
mm	Minor version number
tt	JAR type *i.e., beta, eval)
bbb	Build number

* each letter represents a digit (0-9)

Figure 2, decision block 154 (which may be downloaded with the applet and thus tailored to it) is responsible in this example for correct interpretation of the versioning information -- which in this example is simply an integer and/or string (but could consist of other types of information in other examples). An example string for "version 1.2 build 345" may be for example, "012000345". When a functional module 111 is saved in cache 106, the version number is appended to the functional module's name to make the new module name.

When comparing archive version numbers for compatibility, the example embodiment can apply, as one example, the particular versioning check shown in Figure 5 to ensure that the two versions are compatible (i.e., in this case, that they are interchangeable). Given version number v2 compared against version number v1 for compatibility, v2 is compatible if and only if various tests shown in Figure 5

all return "true". Thus, for example, the example embodiment tests to ensure that the major version number or other identifier match (decision block 260), that the minor version number or other identifier match (decision block 262), that the archive type matches (decision block 264), and that the build number also matches (decision block 266). If all of these tests return "true", then the two functional modules 111 are compatible. If any of the tests return false, then the two functional modules are incompatible and a new version of the functional module 111 should be obtained from server 102. Of course, the Figure 5 process is just an example; other versioning checking processes (which may be downloaded with applet 110) could be used instead to suit individual application requirements.

To optimize download time and functional module 111 processing, the example embodiment uses different formats for downloading functional module 111 packages from server 102 as opposed to storing functional modules in local cache 106. An example download format of the example embodiment is shown in Figure

6. This example download format 300 includes:

- a stream ID field 302 (e.g., a 32-bit integer that serves as both a stream identifier and version number);
- a module version field 304 (the version of the functional module 111 contained within the stream -- see discussion above);
- a signature ID field 306 (an identifier for the signature used for this stream);
- an uncompressed module field 308 (the actual functional module 111 package to be loaded and persisted -- in the example embodiment, all files within the functional module 111 are stored uncompressed); and
- a signature field 310 (the digitally signed digest for this stream 300).

The Figure 6 format allows for optimal compression of the functional module 111 by compressing the entire functional module 111 instead of individual entries within the functional module 111. This means decompression only needs to happen once, at download; and allows downloaded functional modules 111 to be stored into local cache 106 in uncompressed format. Since functional modules 111 loaded from cache 106 are not compressed, they will load more quickly.

In this example, digital signature 310 applies to the compressed stream and not the uncompressed JAR 308; signing the compressed stream allows verification to proceed more quickly because fewer bytes need to be verified upon download from server 102.

In the example embodiment, each downloaded functional module 111 loaded via module loader 200 contains a version file at its root. Module loader 200 uses this file to determine the functional module 111's version and its dependencies. In the example embodiment, the syntax of the file name is:

<module name>.<version #>

(ex: ev3270.012000345 = ev3270.module version 1.2 build 345)

The actual contents of the file provide functional module 111 dependency information. An example format of this file is:

"dependency_list_version" "<version>"

for each dependent module:

"<module name >", "<version>"

When an functional module 111 is loaded and before returning from the "load Module ()" call, module loader 200 parses the dependency file and attempts to load all entries.

The example embodiment thus provides an applet persistence mechanism that is consistent across many supported platforms; provides for the ability to download applets in a piecemeal fashion with control over the granularity of packet size and while minimizing initial download size; and allows incremental changes to applet modules. Preliminary tests have demonstrated that the technique can work on the most popularly available current web browsers (e.g., Netscape 4.5x, Internet Explorer 4.0/5.0 (later build VMs) on Windows NT/98, as well as MRJ and the Java Plug In v2. However, Java 1.1 requires that defining and enforcing security policies is the job of the container and its security manager. If for a particular web browser it is not possible to force the security manager into allowing the example embodiment to create a class and assign privileges, then the example embodiment will not work with that particular platform. In addition, there is always the possibility that a Java applet container vendor may view this technique as a security risk, and tighten the reins on the security manager. However, in JDK2, this risk becomes a non-issue since the security policies allow for the creation of a class loader and applets. In Netscape products, however, creating a class loader is prohibited -- and trying to create one without explicit permission leads to a security exception. The work around seems to be to enable (programmatically) an undocumented security target. This target is currently being used by Netscape to implement Marimba channels in Marimba's NetCaster product, and is thus likely to exist for some time.

While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiments, it is to be understood that the invention is not to be limited to the disclosed embodiment, but

on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.

WE CLAIM:

1 1. A method of distributing software from a server to a client over a
2 computer network, said method comprising:
3 downloading, from said server to said client, an applet including a caching
4 and class loading mechanism; and
5 using said caching and class loading mechanism to retrieve and load
6 additional applet modules on demand.

1 2. A method as in claim 1 wherein said client provides a particular applet
2 execution environment, said downloading step comprises downloading an applet
3 package that has been customized for said particular applet execution environment,
4 and said using step includes retrieving and loading additional applet modules not
5 specific to said particular applet execution environment.

1 3. A method as in claim 1 wherein said client provides a particular web
2 browser, said downloading step comprises downloading an applet package that has
3 been customized for said particular web browser, and said using step includes
4 retrieving and loading additional applet modules not specific to said particular web
5 browser.

1 4. A method as in claim 1 wherein said downloading step includes packaging
2 said caching and class loading mechanism within a stream and downloading said
3 stream in a single http transaction.

1 5. A method as in claim 4 wherein said stream includes an uncompressed
2 archive file containing said caching and class loading mechanism.

1 6. A method as in claim 5 wherein said stream includes at least one digital
2 signature.

1 7. A method as in claim 1 wherein said using step includes verifying, under
2 control of code within said applet, at least one further applet module downloaded
3 on demand.

1 8. A method as in claim 1 further including constructing a cache map with
2 said caching and class loading mechanism.

1 9. A method as in claim 1 wherein said using step includes loading at least
2 further applet module to a persistent cache for subsequent use.

1 10. A method as in claim 1 wherein said additional applet modules are
2 organized into plural functional modules, and said using step comprises
3 downloading at least one of said functional modules on demand.

1 11. A method as in claim 1 wherein said client includes a local non-volatile
2 memory that persistently caches applet classes, and said using step includes
3 requesting an applet class from said server conditioned on determining whether said
4 applet class is already available in said non-volatile memory.

1 12. A method as in claim 1 wherein said client includes a local non-volatile
2 memory, and said using step includes determining whether a particular version of an
3 applet class is stored in said non-volatile memory, and requesting said version from
4 said server based on results of said determining step.

1 13. A method as in claim 1 wherein said using step includes persistently
2 storing applet modules downloaded from said server to said client in a local non-
3 volatile memory associated with said client.

1 14. A system for distributing software from a server to a client over a
2 computer network, said system including a downloader that downloads, from said
3 server to said client over said computer network, an applet including a caching and
4 class loading mechanism, wherein said caching and class loading mechanism
5 executes on said client to retrieve and load additional applet modules on demand.

1 15. A system as in claim 14 wherein said client provides a particular applet
2 execution environment, said downloader downloads an applet package that has
3 been customized for said particular applet execution environment, and said caching
4 and loading mechanism retrieves and loads additional applet modules not specific
5 to said particular applet execution environment.

1 16. A system as in claim 14 wherein said client provides a particular web
2 browser, said downloader downloads an applet package that has been customized
3 for said particular web browser, and said caching and loading mechanism retrieves
4 and loads additional applet modules not specific to said particular web browser.

1 17. A system as in claim 14 wherein said downloader packages said caching
2 and class loading mechanism within a stream and downloads said stream within a
3 single http transaction.

1 18. A system as in claim 17 wherein said stream includes an uncompressed
2 archive file containing said code representing caching and class loading mechanism.

1 19. A system as in claim 18 wherein said stream includes at least one digital
2 signature.

1 20. A system as in claim 14 wherein said caching and loading mechanism
2 verifies at least one further applet class downloaded on demand.

1 21. A system as in claim 14 wherein said caching and loading mechanism
2 maintains a cache map.

1 22. A system as in claim 14 wherein said client includes a persistent cache,
2 and said caching and loading mechanism loads at least further applet class to said
3 persistent cache for subsequent use.

1 23. A system as in claim 14 wherein said additional applet classes are
2 organized into functional modules, and said using caching and loading mechanism
3 downloads said functional modules on demand.

1 24. A system as in claim 14 wherein said client includes a local non-volatile
2 memory that persistently caches applet classes, and said caching and downloading
3 mechanism requests an applet class from said server conditioned on determining
4 whether said applet class is already available in said non-volatile memory.

1 25. A system as in claim 14 wherein said client includes a local non-volatile
2 memory, and said caching and loading mechanism determines whether a particular
3 version of an applet class is stored in said non-volatile memory, and requests said
4 version from said server based on results of said determining step.

26. A system as in claim 14 wherein said client includes a local non-volatile memory associated therewith, and said caching and loading mechanism persistently stores applet classes downloaded from said server to said client.

27. An applet comprising:
a first stream including a platform-dependent caching mechanism; and
at least one further stream including at least one platform-independent functional module,

wherein said platform-dependent caching mechanism is used to cache and load said platform-independent functional module.

28. A method of downloading an applet comprising:
(a) downloading a first stream including a platform-dependent caching and loading mechanism; and
(b) downloading and caching, using said platform-dependent caching and loading mechanism downloaded by step (b), at least one further stream including at least one platform-independent functional module.

29. A system for downloading an applet comprising an applet environment that receives, verifies and begins executing a first stream including a platform-dependent caching and loading mechanism, wherein said applet environment, under control of said platform-dependent caching and loading mechanism, requests, receives and persistently caches at least one further stream including at least one platform-independent functional module.

1 30. An applet-based execution model comprising:
2 initially executing downloaded applet classes that maintain a persistent cache
3 and request additional applet classes that are not in the persistent cache; and
4 subsequently executing said additional applet classes downloaded in
5 response to said additional applet class requests.

1 31. The applet-based execution model of claim 30 wherein said initially
2 executing step includes conditioning said requests on whether said additional applet
3 classes are in said persistent cache.

ABSTRACT OF THE DISCLOSURE

An applet is used to deliver a persistent caching mechanism. An initial package of the applet incorporates a lightweight caching mechanism into its root set of classes. The remainder of the applet are divided into functional modules that can be subsequently downloaded as needed. The initial applet and caching mechanism is packaged and signed in a package using tools and procedures native to each platform; and each functional module is packaged and signed in a generic, platform-independent fashion for verification and loading by any of the various platform-dependent initial applet packages. In this way, the same functional module portions of the applet (which may comprise the bulk of the applet) are cached and loaded in a platform-independent manner without requiring or relying on any caching mechanism built into a platform such as a web browser or other applet execution environment.

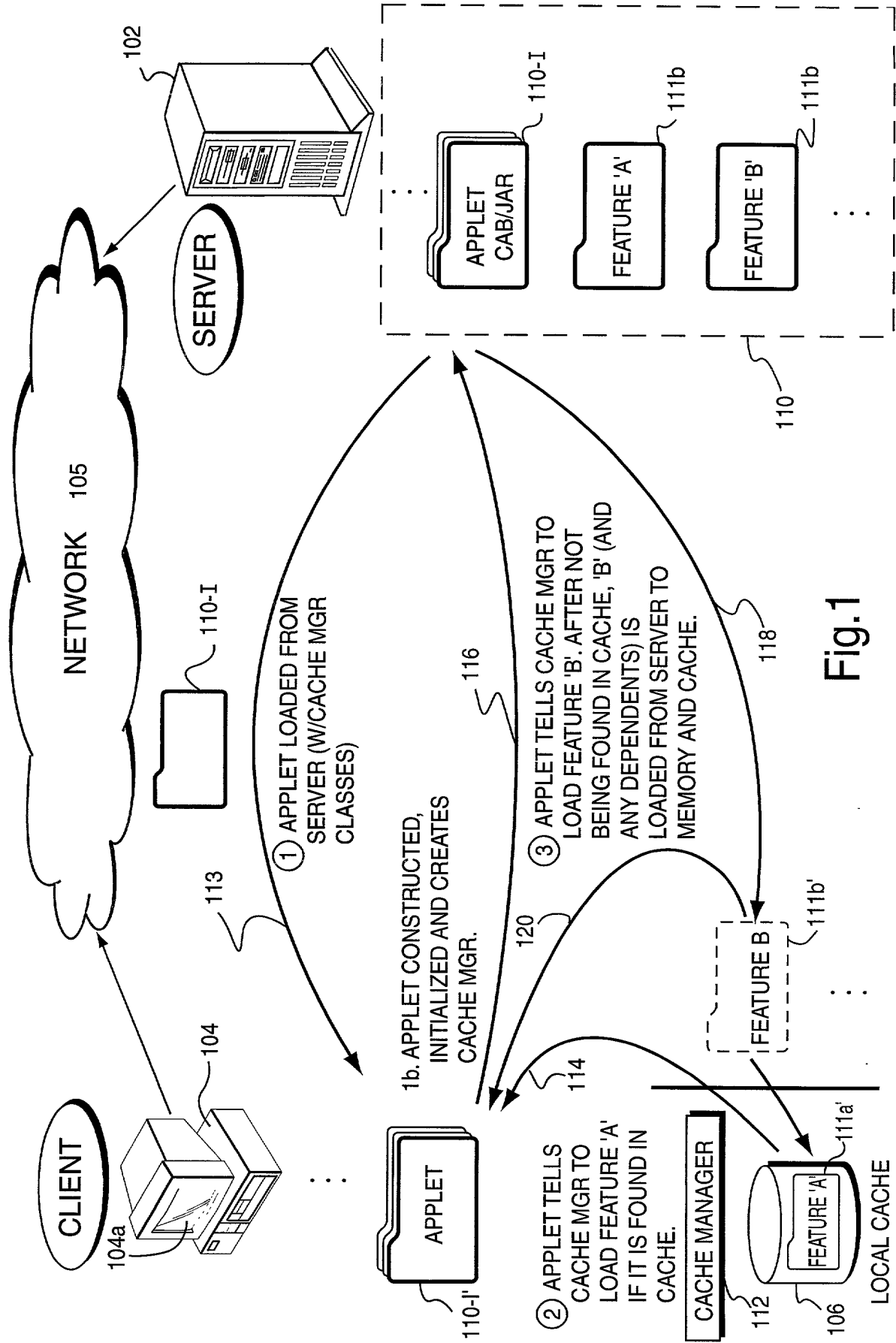
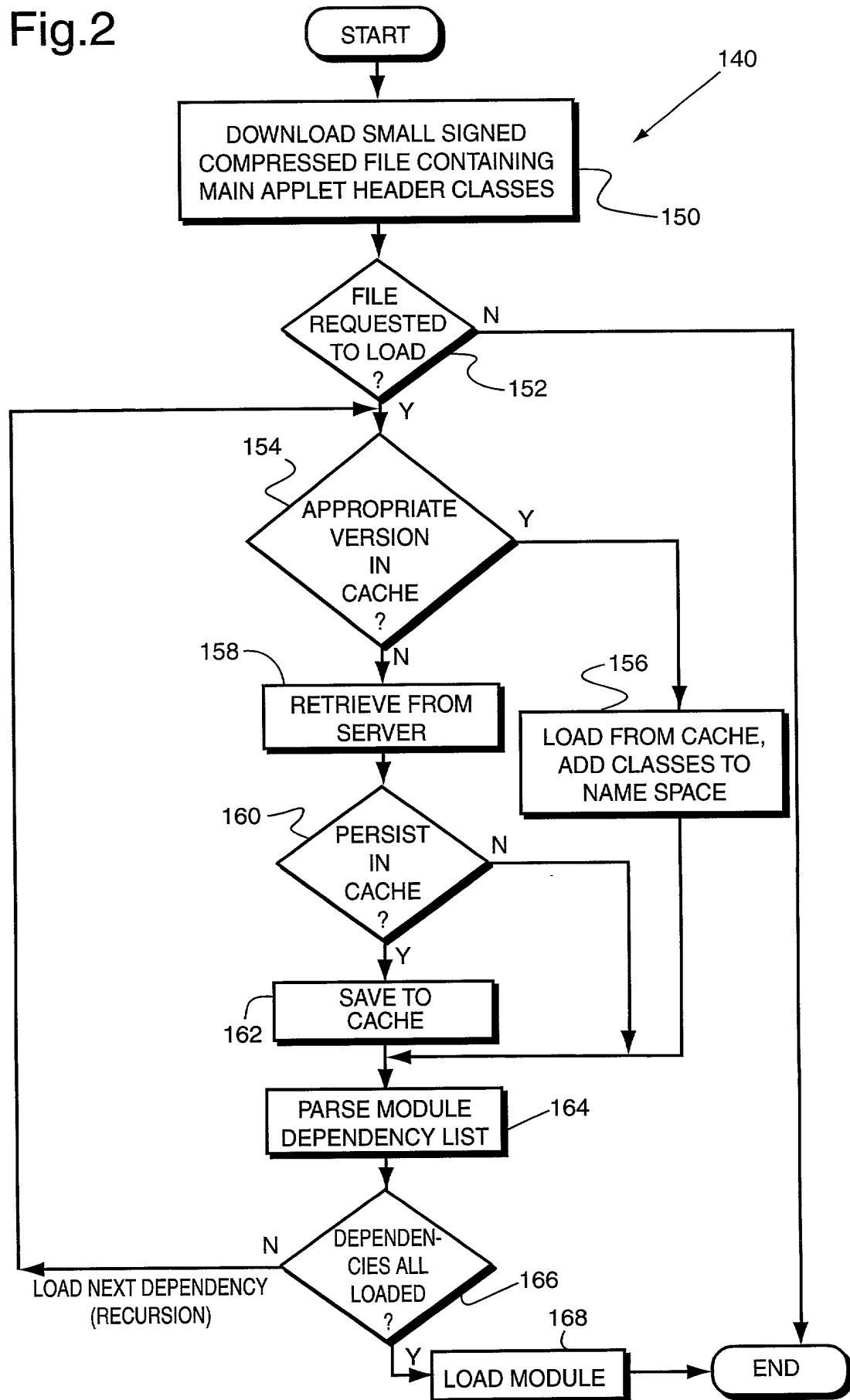


Fig.1

Fig.2



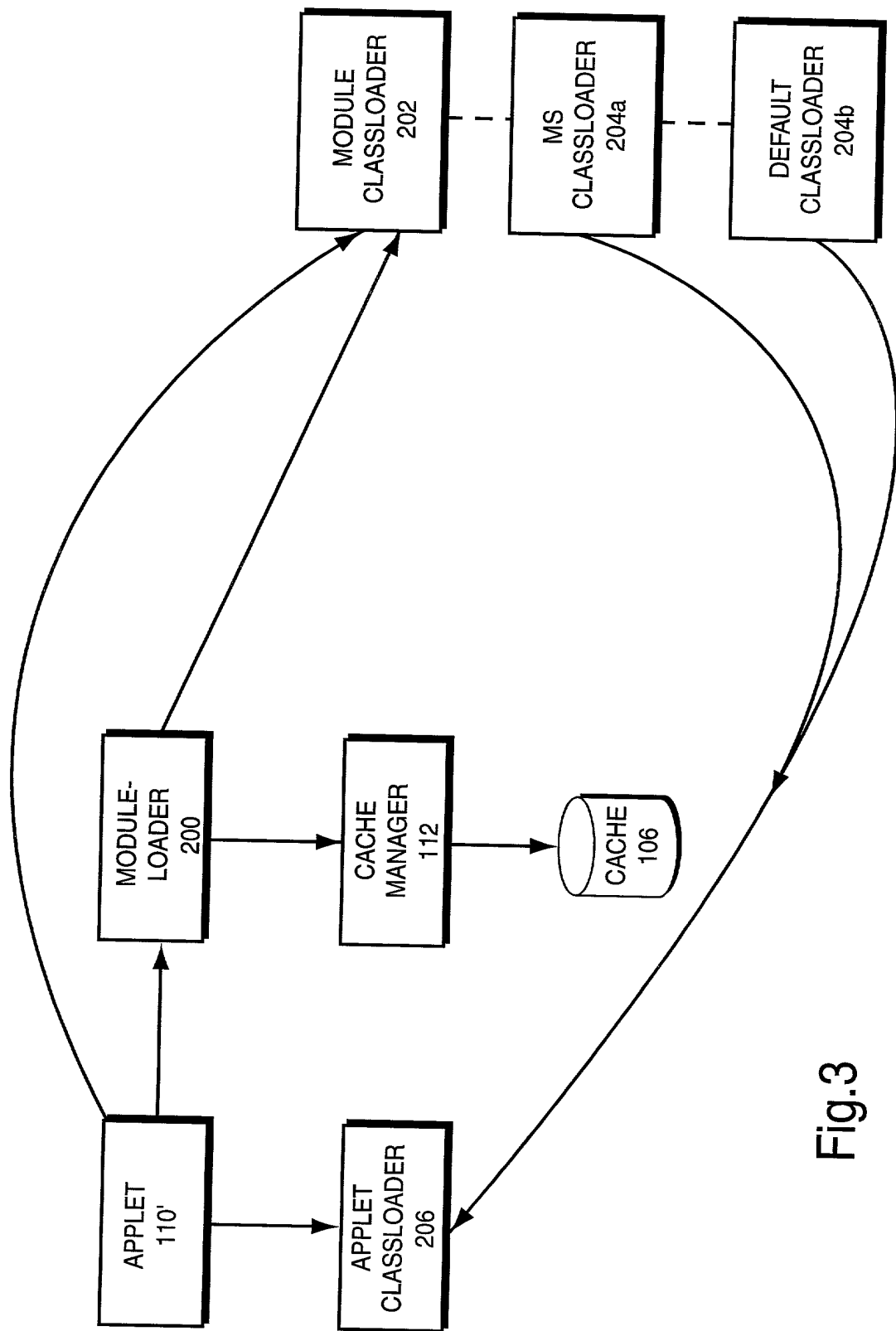


Fig.3

V1 = REQUESTED
V2 = CACHED

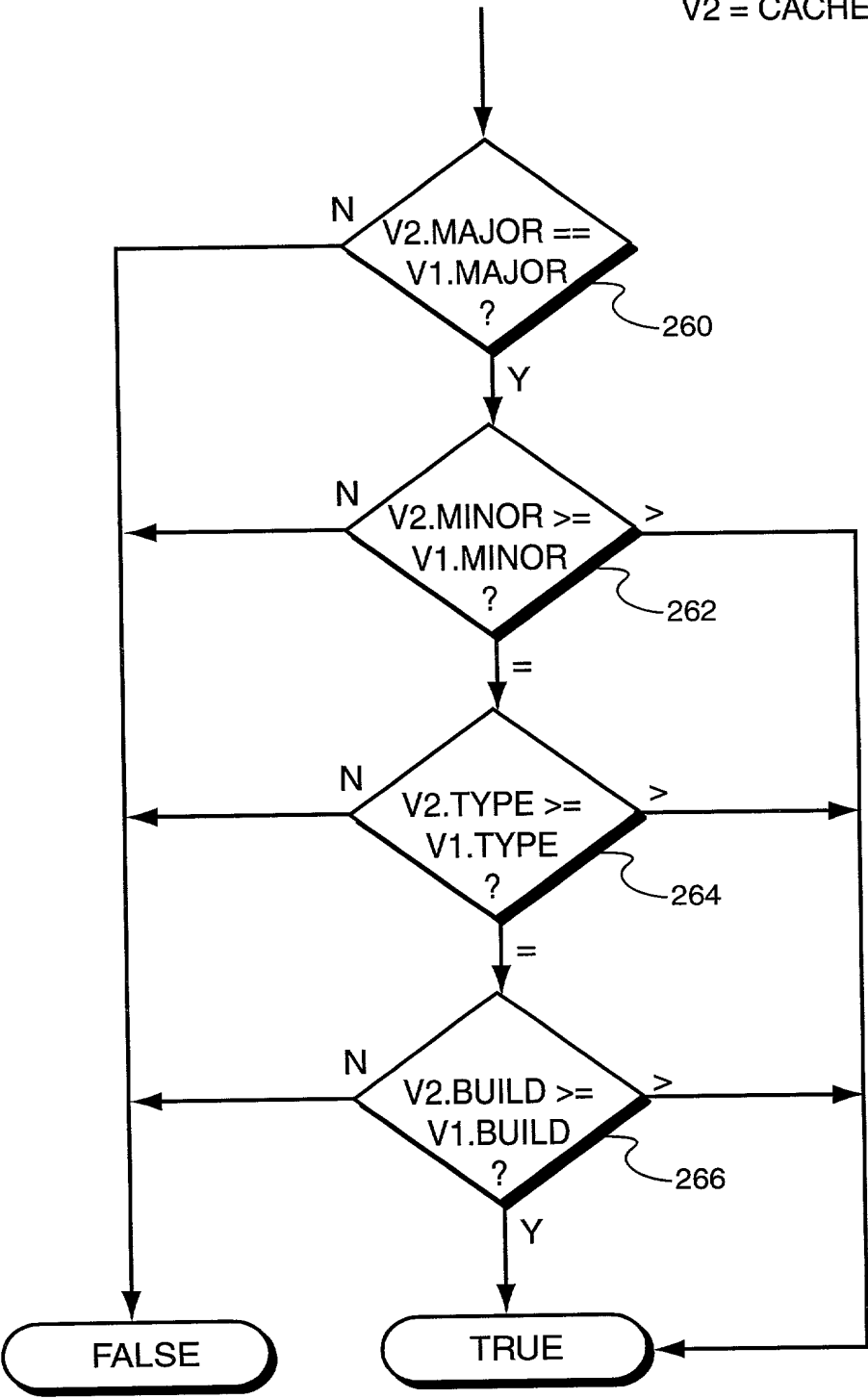


Fig.5

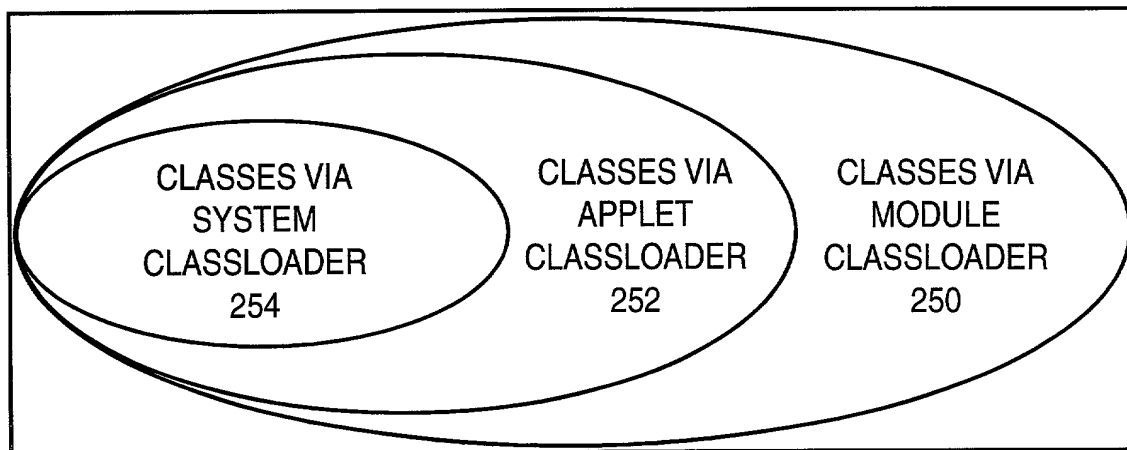


Fig.4

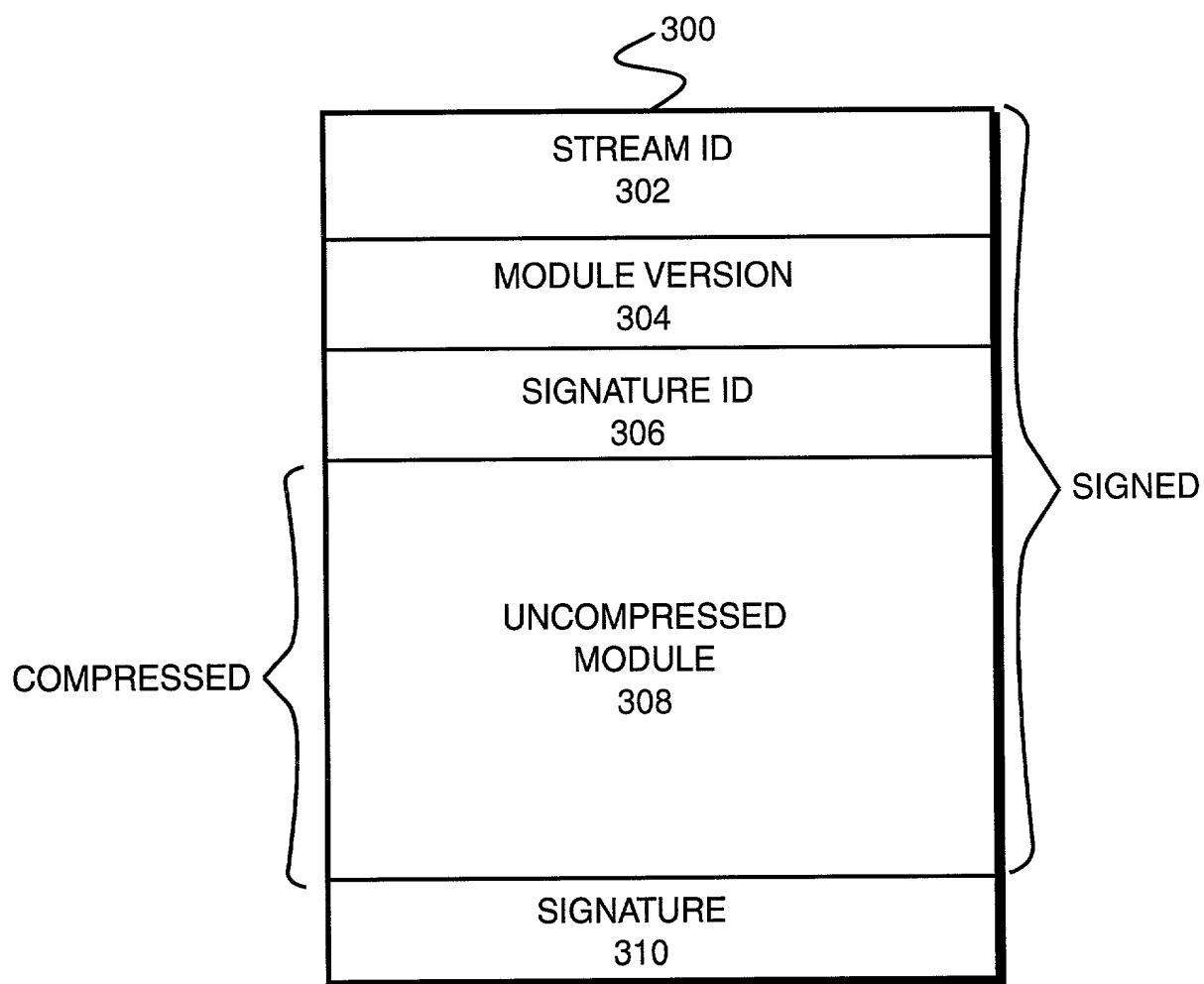


Fig.6

RULE 63 (37 C.F.R. 1.63)
INVENTORS DECLARATION FOR PATENT APPLICATION
IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

As a below named inventor, I hereby declare that my residence, post office address and citizenship are as stated below next to my name, and I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

APPLET EMBEDDED CROSS-PLATFORM CACHING

the specification of which (check applicable box(es)):

- ☒ is attached hereto
☐ was filed on _____

as U.S. Application Serial
No. _____

(Atty Dkt. No. **2452-13**)

☐ was filed as PCT International application No. _____ on _____

and (if applicable to U.S. or PCT application) was amended on _____

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above. I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with 37 C.F.R. 1.56. I hereby claim foreign priority benefits under 35 U.S.C. 119/365 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed or, if no priority is claimed, before the filing date of this application:

Priority Foreign Application(s):

Application Number	Country	Day/Month/Year Filed

I hereby claim the benefit under 35 U.S.C. §119(e) of any United States provisional application(s) listed below.

Application Number	Date/Month/Year Filed

I hereby claim the benefit under 35 U.S.C. 120/365 of all prior United States and PCT international applications listed above or below and, insofar as the subject matter of each of the claims of this application is not disclosed in such prior applications in the manner provided by the first paragraph of 35 U.S.C. 112, I acknowledge the duty to disclose material information as defined in 37 C.F.R. 1.56 which occurred between the filing date of the prior applications and the national or PCT international filing date of this application:

Prior U.S./PCT Application(s):

Application Serial No.	Day/Month/Year Filed	Status: patented pending, abandoned

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon. And on behalf of the owner(s) hereof, I hereby ask that all communications in this matter be directed to:

NIXON & VANDERHYE P.C., 1100 North Glebe Rd., 8th Floor, Arlington, VA 22201-4714, telephone number (703) 816-4000.

1. Inventor's Signature: Craig D. Crutcher Date: 1/14/00
 Inventor: Craig Crutcher U.S.A.
 (first) MI (last) (citizenship)

Residence: (city) Seattle (state/country) Washington

Post Office Address: 507 North 75th Street, Seattle, Washington
 (Zip Code) 98103

2. Inventor's Signature: _____ Date: _____
 Inventor: _____
 (first) _____ MI (last) _____ (citizenship)

Residence: (city) _____ (state/country) _____

Post Office Address: _____
 (Zip Code) _____

